Effectiveness of Combined Sharing and Freeness Analysis using Abstract Interpretation

M.J. García de la Banda
maria@fi.upm.es

M. Hermenegildo
herme@fi.upm.es *or* herme@cs.utexas.edu

Universidad Politécnica de Madrid (UPM)
Facultad de Informática
28660-Boadilla del Monte, Madrid - Spain

Abstract

This paper presents improved unification algorithms, an implementation, and an analysis of the effectiveness of an abstract interpreter based on the *sharing + freeness* domain presented in a previous paper, which was designed to accurately and concisely represent combined freeness and sharing information for program variables. We first briefly review this domain and the unification algorithms previously proposed. We then improve these algorithms and correct them to deal with some cases which were not well analyzed previously, illustrating the improvement with an example. We then present the implementation of the improved algorithm and evaluate its performance by comparing the effectiveness of the information inferred to that of other interpreters available to us for an application (program parallelization) that is common to all these interpreters. All these systems have been embedded in a real parallelizing compiler. Effectiveness of the analysis is measured in terms of actual final performance of the system: i.e. in terms of the actual speedups obtained. The results show good performance for the combined domain in that it improves the accuracy of both types of information and also in that the analyzer using the combined domain is more effective in the application than any of the other analyzers it is compared to.

1 Introduction

The *abstract interpretation* framework [5] provides the basis for a semantics-based approach to dataflow analysis. In the context of Logic Programming Languages, the use of abstract interpretation has been proposed for obtaining characteristics of the program and several types of high level optimizations at compile-time: mode inference analysis [7, 20], efficient backtracking [4], garbage collection [18], aliasing analysis [21, 17, 23], type inference [19, 2], program transformation [8, 9], etc. However, only a few applications of these studies have been reported [27, 26, 19, 29] which provide actual data on the practicality and effectiveness of the inferred dataflow information in the task it was designed for. In particular, there is no report on the effectiveness of a number of approaches proposed for program parallelization in terms of actual speedups obtained.

This paper is aimed at filling the above mentioned gap. It reports on the improvement, implementation, and integration in a real parallelizing compiler of an abstract interpreter based on the *sharing + freeness* domain presented in [23], which was designed to accurately and concisely represent combined freeness and sharing information for program variables. Both the accuracy of the information gathered by the interpreter and its effectiveness are evaluated during its use in the actual task

of automatic parallelization of logic programs and while the interpreter is embedded in a real parallel logic programming system: &-Prolog [10]. These parameters are evaluated in terms of ultimate performance, i.e. the speedup obtained with respect to the sequential version of the program.

The rest of the paper proceeds as follows: we first briefly review the domain and the unification algorithms of [23]. We then improve this algorithm and correct it to deal with some cases which were not well analyzed by the previous algorithm. This is illustrated with an example. We also present abstract unification and all other domain-specific functions for an abstract interpreter implementing the improved algorithm. These functions are also illustrated with an example. We then provide an overview of the evaluation environment and techniques used, as well as the benchmarks. Finally, we present static and dynamic results obtained from the implementation of the system, analyze the effectiveness of the improved algorithm, and discuss the results while comparing them to those obtained from other interpreters available to us.

2 Overview of the Sharing + Freeness Analyzer

Although a detailed description of the *sharing + freeness* analyzer is outside the scope of this paper (and can be found in [23, 24]), in order to put in context the modifications to the original definition of the abstract unification functions and the accuracy and effectiveness of the information inferred by its implementation we will briefly summarize its abstract domain, its applications, and its abstract unification functions in the following subsections.

2.1 Abstract Domain and Applications

The sharing + freeness analyzer was proposed with the objective of obtaining at compile-time accurate variable *groundness*, *sharing*, and *freeness* information for a program and a given query form, i.e., respectively, information on when a program variable will be bound to a ground term, when a set of program variables will be bound to terms which do not share (i.e. they are independent in the sense that they have no variables in common), and when a program variable will be unbound or bound to another variable instead of to a complex term, for the set of queries represented by the query form.

The abstract domain approximates this information by combining two components: one (referred to simply as the "sharing domain" in the rest of the paper) provides information on sharing (aliasing, independence) and groundness; the other encodes freeness information. Informally, the independence and groundness information is represented by the "sharing" domain as follows: an abstract substitution λ for a clause C is a set of sets of program variables ($Pvar$) in that clause which approximates all concrete substitutions θ such that:

- if a program variable X does not appear in any subset of λ, X is bound to a ground term under θ, i.e., $var(X\theta) = \emptyset$.
- if two program variables X and Y do not appear together in any subset of λ, the terms to which those variables will be bound under θ do not share, i.e., $var(X\theta) \cap var(Y\theta) = \emptyset$.
- if two program variables X and Y appear together in at least one subset of λ, nothing can be said about the sharing of the terms to which those variables will be bound under θ, i.e those terms could possibly share under θ.

The component intuitively described above, is essentially the abstract domain of Jacobs and Langen [17]. For efficiency and increased precision, however, the analyzer under study uses the efficient abstract unification and top-down driven abstract interpretation algorithms defined by Muthukumar and Hermenegildo [24] instead of the *pure bottom-up* approach used by Jacobs and Langen.

The freeness domain is represented as a list of those program variables which are known to be free. Its interpretation is the following:

- if a program variable X appears in the freeness component of λ, X is bound to a free variable under θ, i.e., $s.t.\ X\theta = Y, Y \in Pvar$.

- if a program variable X does not appear in the freeness component of λ, but it appears in at least one subset of the sharing component of λ, nothing can be said about the instantiation state of the term to which X is bound under θ, i.e., it can be free, ground or any complex term.

Keeping track of variable sharing is not only required in many types of analysis to ensure correctness, but is also quite useful in a number of applications and, in particular, essential in the compile-time detection of *strict independence* among goals [11], a condition which allows efficient parallelization of programs within the independent and-parallelism model. Informally, this condition states that a set of goals can run in parallel if they do not share any variable at run-time. Freeness information itself is also useful in a number of applications (for example mode generation, program specialization, etc.) and essential in the detection of *non-strict independence* [12] among goals, a condition which extends strict independence, thus enlarging the number of goals which can be run in parallel.

Furthermore, and as expected, the results given and discussed in section 5 show that more accurate information is achieved in each of the domains by allowing communication between the two domains at some points of the analysis: on one hand assuming sharing among arguments of a complex term T can be avoided if the variable which is going to be unified with T is known to be free (information which cannot be inferred if only sharing is abstracted), on the other hand more accurate freeness propagation can be performed if accurate sharing information is provided. The overall effect is thus a more precise analysis than if two separate analyses were performed.

2.2 Abstract Unification Functions

The analyzer is based on the abstract interpretation framework of Bruynooghe described in [1], and adds a number of optimizations. In particular, it uses a specialized fixpoint algorithm [24] which, among other optimizations, includes avoiding recalculation and uses approximations as seeds for convergence improvement. These optimizations reduce the number of iterations needed to reach the fixpoint thus improving efficiency.

Abstract unification is handled by two specialized domain-dependent functions, the *call_to_entry* and *exit_to_success* functions. In the rest of this subsection we will give an intuitive and algorithmic idea of the definition of these functions as given in [23] and we will provide a somewhat more detailed overview of the *call_to_entry* function as background for next section in which the improvements and modifications to this algorithm will be described. Unless otherwise noted, all substitutions referred to in the rest of this section are *abstract* substitutions. It is important to note that although the freeness component has been described as the set of known free variables, for reasons of simplicity in the definition of the functions it will be handled as a mapping from all program variables to the three values F (free), G (ground) and NF (any).

The *call_to_entry* function starts from a subgoal Sg and the calling abstract substitution (λ_{call}), which is the abstract substitution inferred just before performing the abstract execution of Sg, i.e. just before calling any head of a clause to be unified with Sg. It computes the abstract substitution (β_{entry}) which will be obtained from the abstract unification of the subgoal Sg with the head of the clause C for this particular λ_{call}. An intuitive description of the actions performed is as follows:

1. The unification equation $Sg = head(C)$ is simplified into a set of irreducible equations E by the function $simplify\_equations$.

2. The set V is built by mapping the freeness values of the variables in Sg from λ_{call} and taking F as the freeness values of all variables in $head(C)$.

3. Starting with E and V, abstract unification is performed using the function abs_unify_sf. It obtains E', the set of equations in E which are not ground, and V', the updated freeness information extended with freeness values (F) for all variables in the body of the clause.

4. Since some program variables might have become ground due to abstract unification, the updated sharing information for variables in Sg (Sg_share) is computed using the function $update\_sharing\_sf$.

5. Using the sharing information in E' and Sg_share, a conservative estimate of the sharing information in β_{entry} is computed by the functions *powerset_of_set_of_sets* and *partition_sf*.

6. Finally, β_{entry} is computed from its two components, β_share_{entry} (which is obtained by pruning β_share so that it agrees with the sharing information in λ_{call}) and $\beta\_freeness_{entry}$.

On the other hand, the *exit_to_success* function starts from the subgoal Sg, the clause C, and both the λ_{call} and the exit abstract substitution (β_{exit}), which is the abstract substitution inferred just after performing the abstract execution of C, i.e. after the abstract execution of the last subgoal in the body of C. Note that then β_{exit} has the abstract information for the variables in C, instead of that for the variables in Sg. This function first transforms this abstract information in order to deal with just the variables in the head of C, and then it transforms again the information in order to deal just with the variables in Sg.

3 Modifications to the Abstract Algorithm for Computing the Abstract Entry Substitution

In this section we determine the problems which can arise in the computation of the abstract entry substitution when doing so for the two domains of *sharing* and *freeness*. As we shall see, the computation sketched out in the previous section (i.e. that of [23]) can lead to inconsistencies between the information contents in the abstract substitutions for each domain.

3.1 Problems in the Previous Definition

Referring to the intuitive description given previously a problem appears in the third step, i.e. in the definition of *abs_unify_sf*. This step performs two important functions: (1) groundness propagation, and (2) freeness propagation, thus obtaining the updated values, once abstract unification has been performed. However, during the computation, there is no sharing information for the variables in the subgoal Sg (which is contained in the sharing component of λ_{call}). This lack of information results in an incomplete groundness propagation. Let us show it with a simple example. Consider the following situation:

The subgoal Sg	$pred(X_1, X_2)$
the head $head(C)$ (of clause C)	$pred(a, Y_1)$
λ_share_{call}	$\{\emptyset, \{X_1, X_2\}\}$
$\lambda\_freeness_{call}$	$\{X_1, X_2\}$

in which the clause is a fact (and therefore $vars(body(C)) = \emptyset$) and the projected substitution of λ_{call} on the variables of Sg is the same substitution. In the following, we illustrate how β_{entry}, the entry substitution for the clause C, is computed given the above information:

1. `Simplification of the unification equations:`
   $simplify\_equations(\{Sg = head(C)\}) = \{X_1 = a, X_2 = Y_1\}$

2. `Mapping the freeness values of each variable:`
   $asubst\_to\_fvalues(\lambda) = \{X_1/F, X_2/F, Y_1/F\}$

3. `Freeness and groundness propagation:`

   - `First iteration`: The freeness value of X_1 is changed from F to G as the equation$X_1 = a$ is considered (this equation is eliminated from E since all variables in it are assigned to ground values). Subsequently, the second equation $X_2 = Y_1$ is considered. This equation affects the freeness values of neither X_2 nor Y_1, and since it has possibly non ground variables (in fact free variables) it is not eliminated from E.

- `Second iteration`: Only one equation remains in E: $X_2 = Y_1$. This equation can not either be eliminated from E, or modify the freeness values of its variables. Therefore, fixpoint is reached, being the final value of $V = \{X_1/G, X_2/F, Y_1/F\}$ and the final value of $E = \{X_2 = Y_1\}$.

4. `Updating sharing information:`
   $Sg\_share = update\_sharing(V, \lambda\_share) = \{\emptyset\}$
5. `Partitioning of the ''connection graph'' and projection on the head variables:`
   $partition\_sf(V, E, Sg\_share) = \{\{X_2, Y_1\}\}$
   $project\_share(head(C), partition\_sf(V, E, Sg\_share)) = \{\{Y_1\}\}$
   $\beta\_share = \{\emptyset, \{Y_1\}\}$
6. `Building the freeness component as a set of free variables:`
   $fvalues\_to\_asubst(V) = \{Y_1\}$
   $\beta\_freeness_{entry} = \{Y_1\}$
7. $\mathcal{P}(Sg, Sg\_share) = \emptyset$, $(\mathcal{P}(Sg, Sg\_share))^* = \emptyset$. $\beta\_share_{entry} = \{\emptyset\}$.

It is clear that the resulting components of the β_{entry} substitution are inconsistent since they determine the variable Y_1 to be both ground and free. As can be seen in the example, the freeness values of the variables remain unchanged after the third step (only the format of this component is changed, not the information), the rest of steps being aimed at obtaining the most accurate sharing information possible. However, the (final) freeness value of Y_1 has been obtained (third step) without sharing information about the variables of Sg, which is contained in the sharing component of λ_{call}. Therefore, it was impossible to infer that the freeness value of the variable X_2 (with which Y_1 will be bound to after abstract unification) directly depends on the freeness value of X_1. Due to this lack of information, when the freeness value of X_1 is changed from F to G in the first iteration, the freeness value of X_2 remains as F, thus resulting in inconsistent information; i.e. although the sharing component is updated (fourth step) with the information contained in the freeness component, the freeness component is not updated with the information contained in the sharing component.

Since the *exit_to_success* function also uses the function *abs_unify_sf*, it could be thought that the same situation might arise during the computation of the updated groundness and freeness information for the variables in the subgoal Sg, once the analysis of the body of C has been completed (β_{exit}). Since the *abs_unify_sf* function does not keep track of the sharing information among the variables of the head of the clause C contained in the sharing component of β_{exit}, inconsistences among the components of the $\lambda_{success}$ might appear. However, this is not possible since: (1) the sharing information in the sharing component of the λ_{call} has been fully assumed in the sharing component of β_{entry} and therefore it has been also assumed in the sharing component of β_{exit}, and (2) while during the computation of β_{entry} groundness can be propagated from the variables of Sg to the variables of $head(C)$ and vice versa, during the computation of $\lambda_{success}$ only the variables of $head(C)$ provide new groundness information (which will be correctly propagated to the variables of Sg due to (1)). The fact that the definition is correct for this case and that this definition is reused in the *exit_to_success* function was probably the cause of the original incorrect formulation of the function.

3.2 Modifying the Algorithm

The modification of the algorithm for computing the abstract entry substitution is based on the definition of a new *abs_unify_entry* function which replaces the *abs_unify* function and the elimination of the *update_sharing* step which was executed after the replaced function (fourth step). The difference in the new function w.r.t. the definition of the *abs_unify* function is that whenever the freeness value of any variable of Sg changes to G the sharing component of λ_{call} will be updated and, subsequently, the freeness values of the variable in Sg which have been affected (by groundness propagation in the previous step) will also be updated.

The *abs_unify_entry* function takes as input V, E, and the projection λ_{share} of the sharing component of λ_{call} on the variables of Sg. It computes $(V', E', \lambda'_{share})$ where V' and E' are the updated values of V and E as a result of abstract unification, and λ'_{share} is the updated value of λ_{share}.

Assume that $E = \{e_1, \ldots, e_n\}$, where each e_i is of the form $X = Y$ or $X = f(t_1, \ldots, t_m)$. This function performs fixpoint computation on the ordered tuple (V, E, λ_{share}). During each iteration, each $e_i, i = 1, \ldots, n$, is visited using the function *aunify_entry*. After all the equations have been visited, a check is performed on whether any freeness value, equation or sharing information has changed during the last iteration. If so, the fixpoint computation is continued, otherwise (V, E, λ_{share}) is returned.

Definition 1 (abs_unify_entry)

$$abs\_unify\_entry(V, E, \lambda_{share}) =$$
$$= \begin{cases} if & aunify\_entry(V, E, \lambda_{share}, \emptyset) = (V, E, \lambda_{share}) \\ & then\ \ (V, E, \lambda_{share}) \\ else & abs\_unify\_entry(V', E', \lambda'_{share}) \\ & were\ (V', E', \lambda'_{share}) = aunify\_entry(V, E, \lambda_{share}, \emptyset) \end{cases} \square$$

The *aunify_entry* function (figure 3.2) has four input parameters: V, E, λ_{share} and E'. V and λ_{share} are the same as in *abs_unify_entry*, and E and E' are sets of normalized unification equations. This function is invoked by the function *abs_unify_entry* with $E' = \emptyset$ and performs *one* iteration (of abstract unification) by visiting each of the equations in E. During each step, it performs the following actions:

- an equation $e_i \in E, e_i \equiv X = Term$ is removed from E.
- the freeness values in V and the sharing information are updated in the following way: if the freeness value of X is G and $X \in vars(Sg)$ or the freeness value of all variables in $Term$ is G and $vars(Term) \subseteq Sg$, then it is necessary to update first λ_{share} and then V. Otherwise, the freeness values are updated as they were in the *aunify* function.
- finally, the e_i will be added to E' if there is at least one variable in e_i with freeness value different from G.

The function *update_freeness* invoked by *aunify_entry* updates the freeness values of the variables of Sg (contained in V) with the sharing information contained in λ_{share}, i.e. it propagates the groundness information contained in λ_{share} to the groundness information contained in V.

Definition 3 (update_freeness)

$$\begin{aligned} update\_freeness(V, \lambda_{share}) \quad = \quad & V - \{X/\_ | X \in vars(Sg), \nexists S \in \lambda_{share}, X \in S\} \\ & \cup \{X/G | X \in vars(Sg), \nexists S \in \lambda_{share}, X \in S\} \end{aligned} \square$$

The following example illustrates how the execution of *abs_unify_entry* (instead of *abs_unify*) during the computation of the entry substitution computes the correct β_{entry} (avoiding of course inconsistences between its components).

3.3 Example

Consider again the previous example. The computation of the entry substitution will now be performed as follows:

1. `Simplification of the unification equations:`
   $simplify\_equations(\{Sg = head(C)\}) = \{X_1 = a, X_2 = Y_1\}$
2. `Mapping the freeness values of each variable:`
   $asubst\_to\_fvalues(\lambda) = \{X_1/F, X_2/F, Y_1/F\}$
3. `Freeness and groundness propagation:`
   - `First iteration:`

Definition 2 (aunify_entry)

$$
\begin{array}{l}
aunify\_entry(V, \{X = Term\} \cup E, \lambda_{share}, E') = \\
\left\{ \begin{array}{ll}
if & (X/G) \in V, X \in vars(Sg) \ then \ aunify\_entry(V - \{(Y/\_) | Y \in vars(Term)\} \cup \\
 & \qquad \{(Y/G) | Y \in vars(Term)\}, E, \lambda_{share}, E') \\
if & (X/G) \in V, X \notin vars(Sg) \ then \ aunify\_entry(V', E, \lambda'_{share}, E') \\
 & were \ \lambda'_{share} = update\_sharing\_sf(vars(Term), \lambda_{share}) \\
 & and \ V' = update\_freeness(V, \lambda'_{share}) \\
if & vars(Term) = \emptyset, X \notin vars(Sg) \ or \ \forall Y \in vars(Term). (Y/G) \in V, Y \in vars(Sg) \\
 & then \ aunify\_entry(V - \{X/\_\} \cup \{X/G\}, E, \lambda_{share}, E') \\
if & vars(Term) = \emptyset, X \in vars(Sg) \ or \ \forall Y \in vars(Term). (Y/G) \in V, Y \notin vars(Sg) \\
 & then \ aunify\_entry(V', E, \lambda'_{share}, E') \\
 & were \ \lambda'_{share} = update\_sharing\_sf(\{X\}, \lambda_{share}) \\
 & and \ V' = update\_freeness(V, \lambda'_{share}) \\
if & Term \equiv Y \ and \ (X/F) \in V \ and \ (Y/F) \in V \\
 & then \ aunify\_entry(V, E, \lambda_{share}, E' \cup \{X = Term\}) \\
if & Term \equiv Y \ and \ ((X/NF) \in V \ or \ (Y/NF) \in V) \\
 & then \ aunify\_entry(V - \{X/\_, Y/\_\} \cup \{X/NF, Y/NF\}, \\
 & \qquad E, \lambda_{share}, E' \cup \{X = Term\}) \\
if & Term \equiv Y \ and \ (X/NF(e)) \in V \ and \ (Y/F) \in V \\
 & then \ aunify\_entry(V - \{Y/F\} \cup \{Y/NF(e)\}, E, \lambda_{share}, E' \cup \{X = Term\}) \\
if & Term \equiv Y \ and \ (X/F) \in V \ and \ (Y/NF(e)) \in V \\
 & then \ aunify\_entry(V - \{X/F\} \cup \{X/NF(e)\}, E, \lambda_{share}, E' \cup \{X = Term\}) \\
if & Term \equiv Y \ and \ (X/NF(e)) \in V \ and \ (Y/NF(e')) \in V \ and \ e \not\equiv e' \\
 & then \ aunify\_entry(V - \{X/NF(e), Y/NF(e')\} \cup \{X/NF, Y/NF\}, \\
 & \qquad E, \lambda_{share}, E' \cup \{X = Term\}) \\
if & Term \equiv Y \ and \ (X/NF(e)) \in V \ and \ (Y/NF(e)) \in V \\
 & then \ aunify\_entry(V, E, \lambda_{share}, E' \cup \{X = Term\}) \\
if & Term \equiv f(t_1, \ldots, t_n) \ and \ (X/F) \in V \\
 & then \ aunify\_entry(V - \{X/F\} \cup \{X/NF(X = Term)\}, \\
 & \qquad E, \lambda_{share}, E' \cup \{X = Term\}) \\
if & Term \equiv f(t_1, \ldots, t_n) \ and \ (X/NF(X = f(t_1, \ldots, t_n)) \in V \\
 & then \ aunify\_entry(V, E, \lambda_{share}, E' \cup \{X = Term\}) \\
if & Term \equiv f(t_1, \ldots, t_n) \ and \ (X/NF(e)) \in V \ \ and \ \ e \not\equiv X = Term \\
 & then \ aunify\_entry(V - \{X/NF(e)\} - \{(Y/\_) | Y \in vars(Term)\} \\
 & \cup \{X/NF\} \cup \{(Y/NF) | Y \in vars(Term)\}, E, , \lambda_{share} E' \cup \{X = Term\}) \\
if & Term \equiv f(t_1, \ldots, t_n) \ and \ (X/NF) \in V \\
 & then \ aunify\_entry(V - \{(Y/\_) | Y \in vars(Term)\} \cup \\
 & \qquad \{(Y/NF) | Y \in vars(Term)\}, E, \lambda_{share}, E' \cup \{X = Term\})
\end{array} \right. \\
aunify\_entry(V, \emptyset, \lambda_{share}, E) = (V, E, \lambda_{share}) \square
\end{array}
$$

Figure 1: *aunify_entry* algorithm

- First equation $X_1 = a$: the freeness value of X_1 is changed from F to G. Since $X_1 \in vars(Sg)$, λ_{share} will be updated obtaining $\lambda'_{share} = \{\emptyset\}$. With this information, the freeness value of X_2 is updated, being also changed from F to G. Finally, the equation is eliminated from E.
- Second equation $X_2 = Y_1$: since the freeness value of X_2 is G, the freeness value of Y_1 changes from F to G. However it is not necessary to update λ_{share} since $Y_1 \notin vars(Sg)$.

The equation is eliminated from E.

- `Second iteration:` since $E = \emptyset$ there is no change either in the freeness values contained in V or in λ_{share}. Therefore, fixpoint has been reached with $V = \{X_1/G, X_2/G, Y_1/G\}$, $E = \emptyset$ and $Sg_{share} = \{\emptyset\}$.

4. `Partitioning of the ''connection graph'' and projection on the head variables:`
   $partition\_sf(V, E, Sg\_share) = \{\emptyset\}$
   $project\_share(head(C), partition\_sf(V, E, Sg\_share)) = \{\emptyset\}$
   $\beta\_share = \{\emptyset\}$

5. `Building the freeness component as a set of free variables:`
   $fvalues\_to\_asubst(V) = \emptyset$
   $\beta\_freeness_{entry} = \emptyset$

6. $\mathcal{P}(Sg, Sg\_share) = \emptyset$, $(\mathcal{P}(Sg, Sg\_share))^* = \emptyset$. $\beta\_share_{entry} = \{\emptyset\}$.

The resulting β_{entry} correctly determines that, after the abstract unification, all variables in C will be bound to ground terms (and there is no inconsistency between its components).

4 Overview of the Evaluation Environment

The abstract interpretation algorithm as described in the previous sections has been implemented and embedded in the &-Prolog system compiler, and its performance has been evaluated in the task the interpreter was designed for: program parallelization. In the following subsections we will give a brief overview of the &-Prolog system in which the tests for evaluating the effectiveness of the analyzer have been performed, describing the tools which have been used, and the evaluation approach.

4.1 The &-Prolog System

The &-Prolog system comprises a parallelizing compiler aimed at uncovering independent and-parallelism and an execution model/run-time system aimed at exploiting such parallelism. The run-time system is based on the Parallel WAM (PWAM) model, an extension of RAP-WAM [14, 15], itself an extension of the Warren Abstract Machine (WAM) [28]. It is a complete Prolog implementation, offering full compatibility with the DECsystem-20/Quintus Prolog ("Edinburgh") standard, plus supporting the &-Prolog language extensions, which will be described shortly. The user interface is the familiar one with an on-line interpreter and compiler. Prolog code is parallelized automatically by the compiler, in a user-transparent way (except for the increase in performance!). Compiler switches (implemented as "prolog flags") determine whether or not code will be parallelized and through which type of analysis. Alternatively the user can provide Prolog code which is annotated with parallel (&-Prolog) constructs –the compiler still checks the user supplied annotations for correctness, and provides the information obtained from global analysis to aid in the dependency analysis task.

The &-Prolog language is a vehicle for expressing and implementing strict and non-strict independent and-parallelism. &-Prolog is essentially Prolog, with the addition of the parallel conjunction operator "`&`" (used in place of "," –comma– when goals are to be executed concurrently)[1], a set of parallelism-related builtins, which includes several types of groundness and independence checks, and synchronization primitives. Combining these primitives with the normal Prolog constructs, such as "`->`" (if-then-else), users can conditionally trigger parallel execution of goals. For syntactic convenience, an additional construct is also provided: the *Conditional Graph Expression (CGE)*. A CGE has the general form

(*i_cond* `=>` *goal*$_1$ `&` *goal*$_2$ `&` ... `&` *goal*$_N$)

where the *goal*$_i$ are either normal Prolog goals or other CGEs and *i_cond* is a condition which, if satisfied, guarantees the mutual independence of the *goal*$_i$s. The operational meaning of the CGE

[1]The backward operational semantics (backtracking) of the "&" construct is conceptually equivalent to standard backtracking except that dependency information is used to economically perform a limited form of intelligent backtracking. See [16, 14] for details.

is "check *i_cond*; if it succeeds, execute the $goal_i$ in parallel, otherwise execute them sequentially." &-Prolog if-then-else expressions and CGEs can be nested in order to create richer execution graphs. *i_cond* can in principle be any &-Prolog goal but is in general either `true` ("unconditional" parallelism) or a conjunction of checks on the groundness or independence of variables appearing in the $goal_i$s.

4.2 Annotators and Analyzers used

There are three different annotators in the &-Prolog system: the CDG, the UDG and the MEL annotator, whose algorithms are defined in [22].

The CDG annotator is based on the CDG algorithm. It seeks to maximize the amount of parallelism available in a clause, without being concerned with the size of the resultant &-Prolog expression. Firstly, a **C**onditional **D**ependency **G**raph (CDG) is built for each clause. A CDG is a directed acyclic graph where the vertices are subgoals and each edge is labeled by a condition. The condition labeling edge (A,B) is the one that needs to be satisfied so that subgoals A and B are *independent* of each other so that they can be executed in parallel.[2] Later, each CDG is transformed into an &-Prolog expression. It is important to note that this may switch the positions of independent goals, which does not affect their semantics or operational behaviour, since the goals are independent.

The UDG annotator is based on the UDG algorithm, which is essentially the same as the CDG algorithm, except that only unconditional parallelism is exploited, i.e., only goals which can be determined to be independent at compile-time are run in parallel, thus avoiding the overhead due to the run-time checks.

The MEL annotator is based in the MEL algorithm, which creates *only* CGEs in its expressions to achieve parallelism. In addition, it preserves the left-to-right order of subgoals in its expressions. Within these constraints, it seeks to maximize the number of goals to be run in parallel within a CGE. The results from the implementation of this algorithm, were reported in [29].

The two abstract interpreters available which will be used in the evaluation are the *sharing + freeness* interpreter object of this study and the *sharing* only interpreter of [24].

4.3 Evaluation Tools

As mentioned before, the aim in this study is to evaluate the usefulness of the information provided by the analysis in the task it was designed for, program parallelization, and using the speedup obtained w.r.t. the sequential program as the ultimate performance measure. This could be done quite simply by running the parallelized programs in parallel and measuring the speedup obtained. However, this speedup is limited by the number of processors in the system. It would be better to be able to determine the speedup for a large number of processors but while still taking into account real execution times for the sequential parts and scheduling overheads. A novel evaluation environment has been devised in order to achieve this.

The &-Prolog system can optionally generate a trace file during an execution. This file is an encoded description of the events that occurred during the execution of a parallelized program. Examples of such events are parallel fork, start goal, finish goal, join, agent busy, etc. Since &-Prolog generates all possible parallel tasks during execution of a parallel program, even if there are only a few processors in the system, all possible parallel program graphs, with their exact execution times, can be be constructed from this data. A tool has been devised and implemented which takes as input a real execution trace file of a parallel program run on the &-Prolog system, and gives as a result a new optimized trace file which corresponds to the *best possible* execution which would have occurred assuming a system with an infinite number of processors. It also provides statistics about the speedup obtained and the number of processors needed to achieve it.

It is important to note that although this "ideal" parallel execution has been computed, it uses as data a *real* trace execution file. Real execution times of sequential segments and all delay times are taken into account (including not only the time spent in creating the agents, distributing the work, etc, but also the interruptions of the operating system, etc.), and therefore it is possible to consider the results as a very good approximation to the *best possible* parallel execution.

[2] The correctness of these conditions has been shown in [11].

4.4 Programs analyzed

Two broad categories of programs were used for the tests: simple programs and larger ones. Program selection within both categories has been performed taking into account the programs used in those studies with which the results of our tests are going to be compared.

Programs in the First Category	
browse	Gabriel benchmarks, by Tep Dobry and Herve Touati.
deriv	It does symbolic differentiation. The expression given is: E+E-E*E/E*E/E where E is the addition of eight subexpressions
hanoi	It solves the Towers of Hanoi problem. The number of discs given is 9
projgeom	Constructs a perfect difference set of order n in increasing order, starting at 0.
qsort	Quick sort algorithm with difference lists. Two lists have been given as input with lengths: 20 and 100
queens	It solves the n-queens problem.
serialize	Takes a list and converts each item to a number which is the position of that item in the sorted list. The list given as input has 25 characters
vmatrix	It multiplies an N by N matrix and an N by 1 matrix. The input is N = 10
Programs in the Second Category	
asm	the SB-Prolog assembler
boyer	The theorem prover kernel in Gabriel Bench., by E. Tick
peephole	the peephole optimizer used in SB-Prolog
read	The public-domain Prolog tokenizer and parser by Richard O'Keefe and D. H. D. Warren

5 Results

In this section we present the results of the implementation of the *sharing + freeness* analyzer defined in [23] and modified as described in section 3. In this study several Prolog programs have been parallelized using the annotators mentioned in section 4.2 and defined in [22]. The aim is to determine the accuracy and effectiveness of the information provided by the analyzer by obtaining the ratio between the amount of parallelism obtained with this inferred information and the amount of parallelism obtained in two other situations: without analysis information and with the information provided by the *sharing* analyzer defined by Muthukumar and Hermenegildo in [21, 24].

5.1 Static Tests

One way to measure the accuracy and effectiveness of the information provided by abstract interpretation-based analyzers is to compare *statically* (i.e. lexically) the degree of parallelism and overhead introduced in a program which has been parallelized using the information of each analyzer and also using no information, i.e. to compare the number of CGEs, number of groundness and independence checks in the CGEs, and number of unconditional CGEs in the parallelized programs.

We have parallelized several programs using the MEL annotator. The annotator and most of the programs are the same as those used in [13] for presenting results for the MA^3 analyzer. It allows us to compare the results obtained with the *sharing + freeness* analyzer not only to the results of the *sharing* analyzer but also to the MA^3 results. However, it is important to note that the MEL annotator has been recently optimized with simple granularity analysis in builtins (which avoids parallelizing goals with short execution time), and also with a side-effect analysis. Therefore the new MEL annotator can create less CGEs than the one used in [13] and thus numbers differ somewhat. However, it is interesting to compare both results since this gives an idea of the progress made in abstract interpretation-based parallelization.

The results are shown in tables 1 and 2. Table 1 shows the results obtained when parallelizing programs with: no information (columns labeled with **w/o a.**), the information provided by the

program	N. of CGEs			N. of checks			N. of uncond. CGEs		
	w/o a.	share	s + f	w/o a.	share	s + f	w/o a.	share	s + f
asm	8	8	8	26 (3.25)	20 (2.5)	12 (1.5)	0 (0.0)	1 (12.5)	3 (37.5)
boyer	3	3	2	9 (3.00)	6 (2.00)	1 (0.50)	0 (0.0)	0 (0.0)	1 (50.0)
browse	5	5	5	12 (2.40)	10 (2.00)	10 (2.00)	0 (0.0)	1 (20.0)	1 (20.0)
peephole	3	3	2	13 (4.33)	9 (3.00)	3 (1.50)	0 (0.0)	1 (33.3)	1 (50.0)
projgeom	2	2	2	4 (2.00)	1 (0.50)	1 (0.50)	0 (0.0)	1 (50.0)	1 (50.0)
queens	3	3	2	9 (3.00)	2 (0.66)	0 (0.00)	0 (0.0)	1 (33.3)	2 (100.0)
serialize	1	1	1	4 (4.00)	4 (4.00)	1 (1.00)	0 (0.0)	0 (0.0)	0 (0.0)
read	5	5	1	15 (3.00)	5 (1.00)	1 (1.00)	0 (0.0)	0 (0.0)	0 (0.0)
vmatrix	3	3	3	10 (3.33)	1 (0.33)	0 (0.00)	0 (0.0)	2 (66.6)	3 (100.0)
deriv	4	4	4	20 (5.00)	4 (1.00)	0 (0.00)	0 (0.0)	0 (0.0)	4 (100.0)
hanoi	1	1	1	4 (4.00)	0 (0.00)	0 (0.00)	0 (0.0)	1 (100.0)	1 (100.0)
qsort	1	1	1	1 (1.00)	0 (0.00)	0 (0.00)	0 (0.0)	1 (100.0)	1 (100.0)

Table 1: Static Results for the Sharing and Sharing + Freeness Analyzers

program	N. of CGEs		N. of checks		N. of uncond. CGEs	
	w/o a.	MA^3	w/o a.	MA^3	w/o a.	MA^3
asm	123	123	201 (1.6)	101 (0.8)	35 (27.6)	58 (47.2)
boyer	10	10	23 (2.3)	16 (1.6)	3 (30.0)	6 (60.0)
browse	9	9	20 (2.2)	5 (0.5)	0 (0.0)	4 (44.4)
peephole	27	27	116 (4.2)	11 (0.4)	0 (0.0)	19 (70.4)
projgeom	4	4	18 (4.5)	4 (1.0)	0 (0.0)	2 (50.0)
queens	7	7	18 (2.5)	3 (0.4)	1 (14.3)	5 (71.4)
read	42	42	93 (2.2)	36 (0.8)	5 (11.9)	25 (59.5)
serialize	3	3	9 (3.0)	2 (0.6)	0 (0.0)	1 (33.3)
vmatrix	3	3	14 (4.7)	2 (0.6)	0 (0.0)	1 (33.3)

Table 2: Static Results for the MA^3 Analyzer

sharing analyzer (columns labeled with **share**) and the information provided by the *sharing + freeness* analyzer (columns labeled with **s + f**). The three main columns have the following meaning:

- **N. of CGEs**: number of CGEs in the whole parallelized program.
- **N. of checks**: number of checks in the whole parallelized program. Each groundness and independence check has been considered as a unit. The three subcolumns also show in parenthesis the ratio between the number of checks and the number of CGEs, i.e. $checks/CGEs$.
- **N. of uncond. CGEs**: number of unconditional CGEs in the whole parallelized program. The three subcolumns also show in parenthesis the ratio between the number of unconditional CGEs and the total number of CGEs, i.e. $un.CGEs/CGEs$.

Table 2 shows the results described in [13]. The layout is basically the same as in table 1, the only difference is that the subcolumns in which the three main columns are subdived show the results without analysis information and with the information provided by the MA^3 analyzer.

5.2 Dynamic Tests

An arguably better way of measuring the effectiveness of the information provided by abstract interpretation-based analyzers is to measure the speedup achieved in the parallel execution time of the program (ideally for an unbounded number of processors) against the sequential program execution time, while using the information provided by such analyzers in the parallelization. This ideal parallel execution time has been obtained using the tools described in section 4.3.

A related type of test has also been described in [25]. This paper presents a high-level simulation study of the amount and characteristics of the or- and (independent) and-parallelism in a wide selection of Prolog programs, from simple benchmarks to medium-sized applications. In that study, simple programs were parallelized by hand and the others were parallelized with the MEL annotator first,

		deriv	hanoi(9)	vmatrix(10)	qsort(20)	qsort(100)	serialize
MEL.	**w/o a.**	0.08 @ 206	3.01 @ 250	0.05 @ 19	0.16 @ 7	0.30 @ 21	0.16 @ 3
	share	13.05 @ 258	41.77 @ 466	2.35 @ 21	1.58 @ 13	2.79 @ 70	0.16 @ 3
	s + f	42.49 @ 248	41.77 @ 466	5.80 @ 28	1.58 @ 13	2.79 @ 70	0.35 @ 3
CDG.	**w/o a.**	0.08 @ 114	5.64 @ 233	0.05 @ 19	0.17 @ 7	0.30 @ 16	0.16 @ 3
	share	14.42 @ 259	41.77 @ 490	2.64 @ 20	1.57 @ 8	2.79 @ 22	0.16 @ 3
	s + f	42.49 @ 256	41.77 @ 490	6.24 @ 23	1.57 @ 8	2.79 @ 22	0.35 @ 3
UDG.	**w/o a.**	1.0 @ 1	1.0 @ 1	1.0 @ 1	1.0 @ 1	1.0 @ 1	1.0 @ 1
	share	1.0 @ 1	41.77 @ 490	1.04 @ 12	1.57 @ 8	2.79 @ 22	1.0 @ 1
	s + f	42.49 @ 256	41.77 @ 490	6.24 @ 23	1.57 @ 8	2.79 @ 22	1.0 @ 1
MEL.	**w/o a.**	0.82 @ 208	17.80 @ 282	1.00 @ 20	0.8 @ 7	1.78 @ 25	0.88 @ 4
	share	23.54 @ 237	41.77 @ 466	3.75 @ 24	1.58 @ 13	2.79 @ 70	0.88 @ 4
	s + f	42.49 @ 248	41.77 @ 466	5.80 @ 28	1.58 @ 13	2.79 @ 70	0.97 @ 4
CDG.	**w/o a.**	0.83 @ 207	21.30 @ 271	1.21 @ 20	1.14 @ 7	1.92 @ 23	0.96 @ 5
	share	28.03 @ 239	41.77 @ 490	4.19 @ 22	1.57 @ 8	2.79 @ 22	0.96 @ 5
	s + f	42.49 @ 256	41.77 @ 490	6.24 @ 23	1.57 @ 8	2.79 @ 22	1.08 @ 4
manual p.		42.49 @ 256	41.77 @ 490	6.24 @ 23	1.57 @ 8	2.79 @ 22	1.09 @ 4
s. K + H		84.5 @ 248	52.3 @ 427	9.06 @ 18	1.56 @ 3	2.8 @ 8	1.08 @ 4

Table 3: Speed up w.r.t. the Sequential Execution of the Sequential Program

		deriv	hanoi(9)	vmatrix(10)	qsort(20)	qsort(100)	serialize
MEL.	**w/o a.**	3.53	56.54	1.76	2.08	3.174	1.09
	share	13.05	64.38	4.03	2.10	3.00	1.09
	s + f	55.28	64.38	8.29	2.10	3.00	1.09
CDG.	**w/o a.**	3.27	58.69	1.70	2.02	3.00	1.09
	share	14.42	64.38	4.52	2.10	2.98	1.09
	s + f	55.28	64.38	8.90	2.10	2.98	1.09
UDG.	**w/o a.**	1.00	1.00	1.00	1.00	1.00	1.00
	share	1.0	64.38	1.49	2.10	2.98	1.09
	s + f	55.28	64.38	8.90	2.10	2.98	1.09
MEL.	**w/o a.**	5.20	52.93	4.01	2.34	3.98	0.88
	share	46.17	64.38	5.63	2.10	2.98	1.25
	s + f	55.28	64.38	8.29	2.10	2.98	1.25
CDG.	**w/o a.**	4.71	52.96	3.84	2.28	3.66	1.23
	share	49.05	64.38	6.29	2.10	2.98	1.23
	s + f	55.28	64.38	8.90	2.10	2.98	1.23
manual p.		55.28	64.38	8.90	2.10	2.98	1.25

Table 4: Speed up w.r.t. the Sequential Execution of the Parallelized Program

and then optimized by hand. The results presented there will be used here as a reference for the maximum parallelization that can be achieved.

The results are presented in tables 3 and 4. Both are divided horizontally in three blocks. The first two differ in the type of groundness and independence checks used during the execution. While the results contained in the first block were obtained with checks defined in Prolog, the results in the second block were obtained with checks implemented in C. The third block shows the results obtained when parallelizing the programs by hand (labeled as **manual p.**), and the corresponding results presented in [25] (labeled as **s. K + H**).

The first block of each table is divided in three main rows labeled as **MEL**, **CDG** and **UDG**, which indicate the annotator used for each test. The second block has only two main rows since the results corresponding to those programs parallelized with the **UDG** annotator are not affected by the implementation of the checks (they have no checks at all). Each main row is also divided in three rows whose labels show the type of analysis used for the parallelization. The labels are the same, and have the same meaning, as in the static test tables.

Table 3 shows the speedup obtained by the parallelized program w.r.t. the sequential execution of the sequential program and the number of processors needed to obtain it (presented after the @ symbol). Table 4 shows the speedup w.r.t. the sequential execution of the parallelized program. The number of processors is the same as in table 3 and therefore it has not been duplicated.

5.3 Discussion of the Results

Looking at the static results presented in table 1, the first point that can be observed is that the number of resulting CGEs can decrease (`boyer`, `peephole`, `queens` and `read`) if the information provided by the *sharing + freeness* analyzer is considered. This is due to the freeness information: ground checks over the variable X can be known to fail if X is known to be free at this point of the execution, therefore eliminating the CGE and executing the goals sequentially without tests.

The second point is the improved accuracy of the information provided by the *sharing + freeness* analyzer: the results obtained with this analyzer are always better than those obtained without analysis, and equal or better than those obtained with the *sharing* analyzer. This confirms that communication between abstract domains during the analysis increases the accuracy of the resulting information.

It can be thought that although the results of table 1 show that sometimes the *sharing + freeness* analyzer is significatively better than the *sharing* analyzer (e.g. the results obtained for *asm* and *peephole*), in the rest only a few checks are eliminated (four in the best case). However, it turns out that eliminating only one check may produce a great difference in the speedup achieved: the dynamic tests show for *vmatrix* a speedup of up to a factor of 6 with only one check of difference.

Comparing these static results with those presented in table 2, the improvement in the abstract interpretation tools is apparent. Perhaps the most surprising result is the big difference between the number of CGEs obtained for the *asm* benchmark in tables 1 and 2. This is due to the recent MEL optimizations. In particular, granularity analysis reduces the number of CGEs from 123 to 99 and side effects analysis optimization reduces it to 8 (due to the heavy use of write built-ins).

Before discussing the results obtained in the dynamic tests, a few points should be made. Firstly, some of the programs used in this test had to be kept small in size. This is due to the fact that they have small granularity and generate a very large number of tasks (in the order of 10^5) and reach the hardware and software limitations of our "ideal speedup" tools. Secondly, since the dynamic tests have been performed with *real* executions, they always include some number of interruptions (due to the Unix Operating System over which the tools are executing) in the parallel execution which do not allow achieving a real *maximum* parallel execution time. Furthermore, these interruptions produce significant variations among the execution times obtained for the same program. Therefore, the results, which have been taken as the minimum of the times obtained in different (10) executions, can lead to somewhat surprising results when execution time is short as for example in `qsort(20)`: while the speedup obtained parallelizing by hand is 1.57, automatic parallelization can achieve 1.58. This is simply due to "noise" in the measurements over real systems.

The results of dynamic tests show the importance of the information provided by the *sharing + freeness* analyzer and its accuracy, since its results are always equal or better than the rest, and sometimes much better. This is particularly evident for the *vmatrix* and *deriv* programs, in which speedup is significatively higher. As mentioned before, the first case is quite interesting since the difference with the information provided by the *sharing* analyzer results in the elimination of only one check.

It is important to note that in most of the cases (all but *serialize*) hand parallelization obtains the same results as the analyzer. It is somewhat surprising that the speedup of the hand parallelization achieved with the tools used herein is somewhat different in the first three programs from the speedup obtained with the simulator described in [25]. The answer can be in the fact that the simulator of [25] takes as time reference the number of head unifications, considering all head unifications of equal cost, rather than actual execution times. Also, the costs of all builtin predicates appearing in the program are considered equal. We can see in the tables that the differences among the speedups are larger in those programs in which the amount of and-parallelism is high and has a balanced and not small granularity (thus maintaining the processors active most of the time).

Another perhaps surprising result is that actual speedup is achieved with parallelizations in which not all checks have been eliminated, even in the case when the checks are written in Prolog (first blocks of each table): the speedups achieved in the parallel execution of *deriv* w.r.t. the sequential execution, and parallelized with the information obtained by the *sharing* analyzer are 13.05 and 14.42 even when four checks have not been eliminated. The same situation appears in the parallel execution of *hanoi* with no analysis information, in which the speedup obtained is of 2.36 and 2.64. This is even

more pronounced when checks written in C are used. This, coupled with the increased accuracy of the new analyzers, makes the UDG algorithm perhaps not as preferable as it had seemed at first glance. The results also confirm the superiority of the CDG annotator comparing to the other two. However, the results are better at the cost of a great number of processors.

6 Conclusions and Future Work

In this paper improvements to the *sharing + freeness* defined in [23] and the implementation of the new algorithm have been presented. The accuracy of the information provided by this analyzer has been shown by measuring the amount of parallelism achieved through the annotation of sequential programs in three different situations: when no information is available, with information provided by the already implemented analyzer based on the *sharing* domain, as defined in [21], and with information provided by the analyzer whose implementation results have been herein presented.

The amount of parallelism has been measured with two different kinds of tests: static and dynamic. The results of the former have been also compared with those presented in [13] for the MA^3 analyzer; the results of the latter with those of the *sharing* domain analyzer. In any case, static and dynamic tests confirm the improved accuracy of the information provided by the analyzer w.r.t. the corresponding information provided by the other two. Thus we obtain significant speedups upon parallelization of benchmark programs done with this information.

The applications of the analyzer presented are not restricted to IAP detection and exploitation. It has also already shown its effectiveness in program transformation [3], error detection, reordering of goals for improving execution, etc. It has also been applied in the field of CLP, in the analysis of Prolog meta-interpreters for constraint solvers [6]. Further work can be done in improving the analyzer's accuracy by extending it with linear terms and also with type information.

References

[1] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[2] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *5th Int. Conf. and Symp. on Logic Prog.*, pages 669–683. MIT Press, August 1988.

[3] F. Bueno and M. Hermenegildo. Towards an Automatic Translation Algorithm from Prolog to the Andorra Kernel Language. In *Proc. of the 1991 GULP Conference on Logic Programming.* Italian Association for Logic Programming, June 1991.

[4] J.-H. Chang and Alvin M. Despain. Semi-Intelligent Backtracking of Prolog Based on Static Data Dependency Analysis. In *International Symposium on Logic Programming*, pages 10–22. IEEE Computer Society, July 1985.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Rec. 4th Acm Symp. on Prin. of Programming Languages*, pages 238–252, 1977.

[6] M. Garcia de la Banda and M. Hermenegildo. Some Considerations on the Compile-Time Analysis of Constraint Logic Programs. Technical report, U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, 1991.

[7] S. K. Debray and D. S. Warren. Detection and Optimization of Functional Computations in Prolog. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 490–505. Imperial College, Springer-Verlag, July 1986.

[8] J. Gallagher and M. Bruynooghe. The Derivation of an Algorithm for Program Specialization. In *1990 International Conference on Logic Programming*, pages 732–746. MIT Press, June 1990.

[9] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6:159–186, 1988.

[10] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[11] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[12] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[13] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 1991. To Appear.

[14] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[15] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[16] M. V. Hermenegildo and R. I. Nasr. Efficient Management of Backtracking in AND-parallelism. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 40–55. Imperial College, Springer-Verlag, July 1986.

[17] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[18] H. Mannila and E. Ukkonen. Flow Analaysis of Prolog Programs. In *4th IEEE Symposium on Logic Programming.* IEEE Computer Society, September 1987.

[19] A. Marien, G. Janssens, A. Mulkers, and M. Bruynooghe. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *Sixth International Conference on Logic Programming*, pages 33–47. MIT Press, June 1989.

[20] C.S. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 463–475. Imperial College, Springer-Verlag, July 1986.

[21] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[22] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[23] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*. MIT Press, June 1991.

[24] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[25] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *1991 International Logic Programming Symposium*. MIT Press, October 1991.

[26] A. Taylor. LIPS on a MIPS: Results from a prolog compiler for a RISC. Technical report, Association for Logic Programming, June 1990.

[27] P. Van Roy and A. M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *Proceedings of the North American Conference on Logic Programming*, pages 501–515. MIT Press, October 1990.

[28] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.

[29] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*. MIT Press, August 1988.